МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Острозька академія»
Навчально-науковий інститут інформаційних технологій та бізнесу
Кафедра інформаційних технологій та аналітики даних

КВАЛІФІКАЦІЙНА РОБОТА

на здобуття освітнього ступеня бакалавра

на тему: «Розробка інформаційної системи централізованого управління контекстом та інструкціями для AI-систем»

Виконав: студент 4 курсу, групи КН-42
першого (бакалаврського) рівня вищої освіти
спеціальності 122 Комп'ютерні науки
освітньо-професійної програми «Комп'ютерні науки»
Літвінчук Віталій Іванович

Керівник: *доктор філософії з прикладної математики, викладач, фахівець-практик*
Богдан Віталійович Красюк

Рецензент: *кандидат технічних наук, доцент, доцент кафедри прикладної математики Донецького національного університету імені Василя Стуса*
Загоруйко Любов Василівна

РОБОТА ДОПУЩЕНА ДО ЗАХИСТУ
Завідувач кафедри інформаційних технологій та аналітики даних
_____ (проф., д.е.н. Кривицька О.Р.)
Протокол № 11 від «20» травня 2026 р.

Острог, 2026

АНОТАЦІЯ

кваліфікаційної роботи

на здобуття освітнього ступеня бакалавра

Тема: *Розробка інформаційної системи централізованого управління контекстом та інструкціями для AI-систем*

Автор: *студент спеціальності 122 Комп'ютерні науки освітньо-професійної програми «Комп'ютерні науки»*

Літвінчук Віталій Іванович

Науковий керівник: *доктор філософії з прикладної математики, викладач, фахівець-практик*

Богдан Віталійович Красюк

Захищена «……»…… 20__ року.

Пояснювальна записка до кваліфікаційної роботи: 50 с., 0 рис., 0 табл., 30 джерел.

Ключові слова: TUI, OPENTUI, REACT 19, BUN, AI-КОНТЕКСТ, CLI, LLM.

Короткий зміст праці:

Роботу присвячено розробці інформаційної системи для централізованого управління контекстом та інструкціями (промптами) для AI-систем. Виявлено проблему фрагментації промптів у сучасній розробці та неефективності традиційних SaaS-рішень для швидкої локальної роботи.

Спроєктовано та реалізовано термінальний інтерфейс користувача (TUI) з використанням інноваційного стеку: середовище виконання Bun, бібліотека React 19 для декларативного рендерингу та ядро OpenTUI. Система підтримує динамічне форматування, редагування Markdown-інструкцій та швидку навігацію з клавіатури.

Отримані результати підтверджують високу продуктивність рішення, ефективне використання системних ресурсів та покращення Developer Experience (DX) при роботі з великими мовними моделями.

ABSTRACT
of the qualification work
for the Bachelor's degree

Topic: *Development of an information system for centralized management of context and instructions for AI systems*

Author: *Student of degree programme 122 Computer Science within the vocational education programme 'Computer Science'*

Litvinchuk Vitalii Ivanovych

Scientific advisor: *PhD in Applied Mathematics, Lecturer, Practising Specialist Bohdan Vitaliyovych Krasiuk*

Defended «……»…… 20__ year.

Explanatory note to the qualification work: 50 p., 0 fig., 0 tabl., 0 app., 30 ref.

Keywords: TUI, OPENTUI, REACT 19, BUN, AI-CONTEXT, CLI, LLM.

Brief summary of the work:

The work is devoted to the development of an information system for centralized management of context and instructions (prompts) for AI systems. The problem of prompt fragmentation in modern development and the inefficiency of traditional SaaS solutions for fast local work has been identified.

A Terminal User Interface (TUI) was designed and implemented using an innovative stack: Bun runtime, React 19 library for declarative rendering, and OpenTUI core. The system supports dynamic formatting, editing of Markdown instructions, and fast keyboard navigation.

The obtained results confirm the high performance of the solution, efficient use of system resources, and improvement of Developer Experience (DX) when working with large language models.

ЗМІСТ

ВСТУП.....5
РОЗДІЛ 1. ЗАГАЛЬНІ ПОЛОЖЕННЯ.....7
1.1. Опис предметного середовища.....7
1.2. Огляд наявних аналогів.....13
1.3. Постановка задачі.....16
Висновки до розділу 1.....19
РОЗДІЛ 2. ІНФОРМАЦІЙНЕ ТА МАТЕМАТИЧНЕ ЗАБЕЗПЕЧЕННЯ.....21
2.1. Аналіз предметної області (вхідні та вихідні дані).....21
2.2. Проектування архітектури системи.....24
2.3. Математичне та алгоритмічне забезпечення.....27
Висновки до розділу 2.....30
РОЗДІЛ 3. ПРОГРАМНЕ ТА ТЕХНІЧНЕ ЗАБЕЗПЕЧЕННЯ.....32
3.1. Засоби розробки.....32
3.2. Вимоги до технічного та програмного забезпечення.....34
3.3. Опис програмної реалізації.....36
3.4. Керівництво користувача.....41
Висновки до розділу 3.....44
ЗАГАЛЬНІ ВИСНОВКИ.....46
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ.....48

ВСТУП

Сучасний етап розвитку інформаційних технологій характеризується стрімким впровадженням систем штучного інтелекту (AI) у щоденну практику розробки програмного забезпечення. Одним із критичних аспектів ефективної взаємодії з великими мовними моделями (LLM) є якість та структурованість вхідних даних - контексту та системних інструкцій (промптів). Проте, у процесі розробки складних AI-агентів розробники часто стикаються з проблемою фрагментації цих інструкцій, відсутністю версійності та складністю їх тестування в різних середовищах.

Актуальність теми проєкту зумовлена необхідністю створення інструментів, що дозволяють централізовано керувати базою знань та інструкцій для AI-систем безпосередньо через консоль розробника. Використання термінальних інтерфейсів (TUI) забезпечує високу швидкість роботи та легку інтеграцію в існуючі CI/CD процеси. На відміну від громіздких веб-інтерфейсів, TUI пропонує швидку навігацію з клавіатури, що значно покращує Developer Experience (DX).

Метою кваліфікаційної роботи є розробка інформаційної системи для централізованого управління контекстом та інструкціями для AI-систем із використанням сучасного стека технологій: середовища виконання Bun, бібліотеки React 19 та фреймворку OpenTUI.

Для досягнення поставленої мети було визначено такі завдання:

– проаналізувати предметне середовище та сформулювати вимоги до системи управління контекстом;

– провести огляд наявних аналогів та обґрунтувати вибір технологічного стека;

– спроєктувати архітектуру CLI-застосунку, що забезпечує реактивне оновлення інтерфейсу в терміналі;

– реалізувати програмний код системи, забезпечивши ефективну обробку користувацького введення та зберігання інструкцій;

– провести тестування розробленого інструменту та оцінити результати його роботи.

Об’єктом дослідження є процеси управління життєвим циклом інструкцій та контекстних даних для AI-систем.

Предметом дослідження є програмне забезпечення (CLI-додаток) для централізованої каталогізації, редагування та передачі контексту в зовнішні сервіси, а також методи його створення з використанням реактивного програмування у терміналі.

РОЗДІЛ 1. ЗАГАЛЬНІ ПОЛОЖЕННЯ

1.1. Опис предметного середовища

Сучасна парадигма розробки програмного забезпечення перебуває у стані глибокої трансформації, спричиненої стрімкою інтеграцією великих мовних моделей (Large Language Models, LLM) в архітектуру корпоративних та користувацьких застосунків. Якщо раніше взаємодія між компонентами системи спиралася на жорстко визначені протоколи (REST, gRPC, GraphQL) та детерміновані формати даних, то сьогодні розробники стикаються з недетермінованим характером AI-агентів. Фундаментальною одиницею комунікації з такими моделями є промпт (системна інструкція) - текстовий або мультимодальний блок інформації, який формує контекст та визначає поведінку моделі.

Щоб глибоко зрозуміти необхідність та складність управління промптами, необхідно детально розглянути еволюцію та архітектуру сучасних мовних моделей. Фундаментом для революції ШІ стала архітектура Transformer, вперше запропонована дослідниками Google у 2017 році в епохальній статті 'Attention Is All You Need'. До появи трансформерів домінуючими підходами в обробці природної мови (NLP) були рекурентні нейронні мережі (RNN) та мережі довгої короткострокової пам'яті (LSTM). Вони обробляли інформацію послідовно, слово за словом. Це створювало фундаментальну проблему 'забування' контексту при обробці довгих речень, оскільки градієнт затухав під час зворотного поширення помилки (Backpropagation) через багато часових кроків.

Трансформери радикально змінили цей підхід, відмовившись від рекурентності на користь механізму уваги (Self-Attention). Механізм Self-Attention дозволяє моделі аналізувати кожне слово у реченні в контексті всіх інших слів одночасно. Замість послідовної обробки, трансформер розраховує три матриці для кожного токена: Запит (Query - Q), Ключ (Key - K) та Значення (Value - V). Математично цей механізм обчислюється як зважена сума матриць значень, де ваги визначаються через функцію Softmax від скалярного добутку матриць запитів і ключів, поділеного на квадратний корінь із розмірності ключів. Це дозволяє моделі

ефективно 'звертати увагу' на релевантні слова незалежно від їхньої фізичної дистанції у реченні.

Важливою складовою цієї архітектури є позиційне кодування (Positional Encoding). Оскільки трансформер обробляє всі слова паралельно (що забезпечує фантастичну швидкість навчання на GPU), він втрачає інформацію про порядок слів. Щоб модель розуміла синтаксис (різницю між 'собака вкусив людину' і 'людина вкусила собаку'), до векторних представлень слів (Embeddings) додаються синусоїдальні та косинусоїдальні хвилі різної частоти. Саме тому порядок та структура блоків інформації у промпті, створюваному розробником, безпосередньо впливає на активацію тих чи інших нейронів у багатошаровому перцептроні (Feed Forward Neural Network), який йде після блоку Self-Attention.

Збільшення обчислювальних потужностей та обсягів тренувальних даних призвело до появи так званих Великих Мовних Моделей (LLM). Еволюція відбувалася шляхом експоненційного масштабування параметрів (ваг) мережі. Від мільйонів параметрів у ранніх моделях індустрія перейшла до мільярдів (наприклад, Llama 2 70B, GPT-3 175B) і навіть трильйонів параметрів в архітектурах Mixture of Experts (MoE). Масштабування відкрило так звані 'емерджентні властивості' (Emergent Abilities) - здібності моделей вирішувати задачі, яким їх безпосередньо не навчали: від написання коду на екзотичних мовах програмування до вирішення складних логічних задач.

Історія розвитку інженерії підказок (Prompt Engineering) демонструє стрімку еволюцію від простих текстових запитів до складних програмованих структур, що супроводжують ці гігантські моделі. У ранніх моделях, таких як GPT-2, взаємодія зводилася до банального продовження тексту. З появою GPT-3 та розширенням контекстних вікон, індустрія відкрила для себе концепцію навчання на кількох прикладах (Few-Shot Prompting). Модель навчилася адаптуватися до нових завдань без необхідності донавчання (Fine-tuning), покладаючись виключно на контекст, переданий у системній інструкції. Це змістило парадигму розробки: замість дорогих і складних тренувань моделей з нуля, розробники зосередилися на вдосконаленні алгоритмів генерації вхідного контексту.

З виходом моделей сімейства GPT-4, Claude 3 Opus та Gemini Ultra, інженерія підказок перетворилася на самостійну науково-практичну дисципліну з глибоким математичним бекграундом. З'явилися такі складні патерни, як Chain-of-Thought (ланцюг міркувань), Tree-of-Thought (дерево міркувань), Graph-of-Thought (граф міркувань) та ReAct (Reasoning and Acting). Наприклад, патерн Chain-of-Thought змушує модель розбивати складну задачу на покрокові етапи міркувань (Step-by-step reasoning) перед генерацією остаточної відповіді. Це емпірично доведено збільшує точність вирішення математичних задач та складних логічних головоломок. Ці патерни вимагають формування багатоетапних, глибоко структурованих інструкцій, які можуть налічувати десятки тисяч токенів і містити складні JSON або XML схеми для парсингу результатів.

Проте, незважаючи на стрімкий розвиток самих мовних моделей, інструментарій розробників для управління цими інструкціями тривалий час залишався на примітивному рівні. Коли системні інструкції для штучного інтелекту розпорошені по вихідному коду, проєкт швидко перетворюється на «археологічні розкопки», де управління залежностями та змінами стає вкрай складним завданням. Дослідження предметного середовища виявляє декілька критичних проблем, які суттєво уповільнюють розробку та знижують надійність AI-застосунків.

По-перше, однією з найпоширеніших проблем є так звана «Hardcoded-пастка». У традиційній розробці винесення конфігурацій та текстових ресурсів за межі виконуваного коду є стандартом де-факто (наприклад, файли .env, локалізація, YAML-конфігурації). Проте в розробці AI-систем інженери часто жорстко кодують масивні багаторядкові промпти безпосередньо всередині TypeScript або Python файлів. Це призводить до ситуації, коли будь-яка зміна (навіть виправлення однієї коми або оптимізація інструкції для кращого JSON-виводу моделі) вимагає проходження повного циклу CI/CD: створення комміту, проходження стадії білда, запуску автоматизованих тестів та розгортання в робочому середовищі. Це створює штучне «пляшкове горло», позбавляючи розробників можливості гнучко експериментувати з налаштуваннями моделі.

По-друге, спостерігається небезпечний феномен дрейфу інструкцій (Prompt Drift). У міру того як проєкт масштабується і до нього залучається більше команд, промпти починають неконтрольовано еволюціонувати в різних напрямках. Наприклад, команда маркетингу та дизайну (Product Team) може вимагати, щоб AI-асистент звучав «максимально дружньо та емпатично», і оновлює інструкції у фронтенд-сервісі. Водночас розробники бекенду або іншого мікросервісу, не знаючи про ці зміни, можуть прописати в системному промпті сувору вимогу «відповідай максимально лаконічно, використовуй технічний стиль». У результаті система як єдине ціле починає демонструвати роздвоєння особистості продукту. Різні частини застосунку використовують несумісні або застарілі версії копірайту, що руйнує цілісний користувацький досвід (User Experience).

Третьою складовою проблематики є криза управління контекстним вікном (Context Window) та форматуванням. Ефективний промпт-інжиніринг - це далеко не просто статичний текст, це складна та динамічна збірка даних, яка формується у реальному часі. Сучасні моделі мають обмеження на кількість вхідних токенів. Хоча новітні моделі підтримують вікна у 128 або навіть 200 тисяч токенів, існує феномен 'Lost in the Middle' - зниження здатності моделі вилучати інформацію з середини величезного контексту. Розробнику потрібно втиснути у жорсткий ліміт токенів лише найрелевантнішу інформацію: результати семантичного пошуку (RAG - Retrieval-Augmented Generation), історію попереднього діалогу, метадані користувача, правила безпеки (Guardrails) тощо. Коли логіка цієї збірки фрагментована по різних класах і функціях, контролювати чистоту контексту стає вкрай важко. Це часто призводить до переповнення контекстного вікна (Context Overflow) та відхилення моделі від заданої теми (Hallucinations).

Додатковою проблемою в контексті управління токенами є алгоритми токенізації, такі як Byte-Pair Encoding (BPE) або WordPiece. Токенізація не має однозначної відповідності з символами або словами; один токен може бути частиною слова, цілим словом або навіть фрагментом коду. Різні моделі використовують різні словники токенів (Vocabularies). Наприклад, токенізатор cl100k_base, який використовується в моделях сімейства GPT-4, розбиває код і текст

інакше, ніж токенізатори сімейства Llama 3. Відсутність централізованого контролю над тим, як генерується промпт, позбавляє розробника можливості здійснювати точний підрахунок токенів та оцінювати фінансові витрати (Cost Estimation) на кожний виклик API ще до його відправлення.

Крім того, синтаксичне пекло також додає значних перешкод розробникам. Ринок LLM наразі є висококонкурентним, і розробники часто використовують кілька моделей від різних провайдерів (OpenAI, Anthropic, Meta) одночасно для різних задач (Model Routing). Однак кожна мовна модель має свої специфічні вподобання щодо форматування вхідних даних. Моделі сімейства GPT-4 чудово розуміють та структурують дані у форматі Markdown. Моделі Claude від Anthropic демонструють найвищу ефективність при використанні XML-тегів для ізоляції різних блоків інструкції. Відкриті моделі сімейства Llama потребують специфічних керуючих токенів початку та кінця інструкції (наприклад, [INST] та [/INST]). Без єдиної системи управління контекстом розробник змушений створювати складні, важко підтримувані логічні конструкції (if-else блоки) для ручного форматування контексту під кожного конкретного провайдера AI.

Нарешті, відсутність версійності (аналогу Git для промптів) ускладнює контроль якості та тестування. У класичному програмуванні зміна детермінованого алгоритму завжди дає передбачуваний результат, який легко покрити модульними тестами. У взаємодії з імовірнісними моделями типу LLM навіть незначна зміна однієї фрази або перестановка речень місцями може кардинально змінити ваги уваги (Attention Weights) всередині нейронної мережі і, як наслідок, неочікувано зламати парсинг вихідного JSON. У таких умовах життєво необхідною є можливість зробити швидкий відкат (rollback) до попередньої версії промпту, яка демонструвала стабільну роботу на етапі тестування. Однак без централізованої системи версіонування та зберігання такий підхід реалізувати практично неможливо.

Ще одним вагомим аргументом на користь розробки локальної системи є аспект кібербезпеки та конфіденційності даних. Управління промптами тісно пов'язане із захистом інтелектуальної власності компанії (Proprietary Instructions). У багатьох випадках системні інструкції містять унікальне бізнес-ноу-хау, яке робить

AI-продукт конкурентоспроможним. Окрім цього, існує загроза так званих ін'єкцій підказок (Prompt Injections), коли зловмисник маніпулює вхідними даними, щоб змусити модель ігнорувати системні інструкції та виконувати неавторизовані дії (Jailbreaking). Відсутність централізованого управління унеможливлює ефективний аудит безпеки промптів та впровадження стандартизованих захисних бар'єрів (Defensive Prompting).

Окремої уваги заслуговує проблема оркестрації AI-агентів (Agent Orchestration). Сучасні архітектурні патерни, такі як Multi-Agent Systems (MAS) та Agentic AI, передбачають використання кількох спеціалізованих AI-агентів, кожен з яких має свій унікальний набір інструкцій та персоналізований контекст. Наприклад, один агент може відповідати за генерацію коду, інший - за його рев'ю, третій - за написання документації. Кожен з цих агентів потребує свого власного системного промпту з чітко визначеною роллю, обмеженнями та форматом відповіді. Без централізованого сховища та ефективного інструменту навігації управління десятками або сотнями таких спеціалізованих інструкцій перетворюється на нездійсненне завдання, що гальмує впровадження складних мультиагентних архітектур у виробничих системах.

Також необхідно враховувати проблему відтворюваності (Reproducibility) експериментів з мовними моделями. У наукових та інженерних дослідженнях критично важливо мати можливість повторити експеримент із тими самими вхідними даними та отримати ідентичний або статистично близький результат. Коли промпти не версіонуються і змінюються ad-hoc, відтворити попередній результат стає неможливо. Це особливо критично при порівнянні продуктивності різних моделей (Benchmarking) або при налагодженні (Debugging) регресій у якості генерації. Система централізованого управління контекстом повинна забезпечувати фіксацію точного стану кожного промпту на момент проведення експерименту, дозволяючи повертатися до будь-якої попередньої версії інструкції.

Нарешті, слід зазначити вплив ергономіки робочого середовища на продуктивність інженера. Дослідження компанії GitHub (State of the Octoverse 2024) показують, що розробники витрачають до 40 відсотків робочого часу на навігацію

між файлами, пошук потрібних фрагментів коду та перемикання між контекстами. Коли до цього додається необхідність шукати промпти у різних сервісах та копіювати їх через буфер обміну, ефективність падає ще більше. Інтеграція управління промптами безпосередньо у робоче середовище терміналу (яке є природним для бекенд-розробників, DevOps-інженерів та дослідників AI) дозволяє мінімізувати ці непродуктивні витрати часу та зберегти стан глибокого фокусу (Deep Work).

1.2. Огляд наявних аналогів

Аналіз сучасного ринку інструментів для розробників AI-систем показує, що існуючі рішення для управління промптами та контекстом (Prompt Management Systems) розподіляються на кілька основних категорій. Еволюція цих інструментів відбувалася паралельно з еволюцією самих мовних моделей, переходячи від простих текстових редакторів до складних екосистем. Проте кожна з існуючих категорій має свої фундаментальні недоліки в контексті швидкої та гнучкої локальної розробки.

Перша категорія - це складні Enterprise-платформи, так звані «важкі» SaaS-рішення (наприклад, LangSmith, PromptLayer, Weights & Biases). Вони пропонують потужні візуальні дашборди, глибоку аналітику, трекінг вартості токенів та можливості для колаборації команд. Ці платформи дозволяють нетехнічним спеціалістам (наприклад, копірайтерам чи менеджерам продуктів) брати участь у налаштуванні промптів. Проте для багатьох команд розробників, особливо на етапах прототипування або при створенні автономних агентів, такі платформи створюють суттєве тертя (friction) у щоденному робочому процесі (Developer Workflow).

Найбільшим недоліком наявних SaaS-рішень є контекстне перемикання (Context Switching). Дослідження у сфері продуктивності розробників свідчать, що перемикання між різними контекстами та застосунками значно знижує фокус та ефективність. Розробнику доводиться постійно виходити зі свого звичного середовища розробки (IDE або терміналу), відкривати браузер, шукати потрібний промпт у складному веб-інтерфейсі платформи, копіювати текст, переносити його в код або робити додатковий мережевий запит до API платформи. Це руйнує стан

«потоку» інженера і збільшує когнітивне навантаження. Сучасний інструмент управління конфігураціями повинен бути органічною частиною екосистеми розробника, дозволяючи витягувати промпти як звичайні локальні залежності, без обов'язкових зовнішніх мережевих викликів під час локального тестування.

Іншим значним недоліком комерційних SaaS-платформ є складність локальної розробки та тестування (Local First Development). Вкрай важко налаштувати ізольоване локальне середовище для AI-агента, якщо його фундаментальна логіка - інструкції - зберігається у хмарному сторонньому сервісі. Без доступу до інтернету розробка повністю зупиняється. Крім того, виникають питання приватності та безпеки даних. Згідно зі стандартами корпоративної безпеки (наприклад, SOC2 або GDPR), компанії часто не бажають передавати свої пропрієтарні інструкції та логи генерацій на сервери третіх сторін. Хмарні платформи змушують компанії або миритися з цими ризиками, або інвестувати величезні кошти у розгортання On-Premise версій цих рішень.

Для подолання цих проблем та усунення так званого «Prompt Spaghetti» (хаотичного нагромадження інструкцій у коді) необхідний підхід Prompt-as-Code. Цей підхід пропагує винесення системних інструкцій у зовнішні локальні версіоновані файли (наприклад, Markdown, YAML або JSON) з можливістю їхньої динамічної ін'єкції у код. Таке рішення дозволяє зберігати інструкції в тому ж Git-репозиторії, що і вихідний код, гарантуючи синхронізацію версій промптів із версіями бекенду. Будь-яка зміна промпту проходить через стандартний процес Code Review у системі контролю версій (GitHub, GitLab), що гарантує прозорість та відстежуваність змін.

Для реалізації підходу Prompt-as-Code існують традиційні інструменти командного рядка (CLI - Command Line Interface). CLI-інструменти (на кшталт grep, awk, sed або спеціалізованих утиліт) дозволяють виконувати швидкі операції з файлами, але вони обмежені парадигмою «запит-відповідь» (Request-Response). CLI виводить текст лінійно (stdout) і завершує роботу процесу. При роботі зі складними багаторядковими промптами, які можуть налічувати сотні рядків тексту, форматувань, XML-тегів та JSON-схем, традиційний CLI стає незручним. Текст

швидко зникає за межами екрану терміналу (у буфері прокрутки), а редагування перетворюється на виклик зовнішніх редакторів типу Vim або Nano, що розриває безперервний процес тестування.

Ефективною альтернативою традиційним CLI є TUI (Terminal User Interface). TUI займає унікальну технологічну нішу між класичним CLI та повноцінним GUI (Graphical User Interface). На відміну від CLI, TUI використовує всю площину термінала (екрану) для створення динамічних панелей, вікон, форм введення та списків, що оновлюються в реальному часі і негайно реагують на натискання клавіш. Водночас TUI повністю позбавлений важкого графічного стека (відсутня потреба у відеокарті, складному DOM-дереві браузера, рушіях Chromium чи віконних менеджерах ОС). TUI працює виключно з текстовими символами та керувальними послідовностями ANSI. Це робить його надзвичайно легким, блискавично швидким, економним до ресурсів персонального комп'ютера (RAM/CPU) та повністю доступним навіть при підключенні до віддалених серверів через протокол SSH. Власне, створення ефективного TUI-додатка для централізованого управління промптами є найбільш раціональним рішенням, яке об'єднує швидкість роботи в терміналі з багатством графічного інтерфейсу, зберігаючи філософію клавіатурної навігації (Mouse-less flow).

Друга категорія аналогів - це фреймворки та бібліотеки для оркестрації LLM-викликів (наприклад, LangChain, LlamaIndex, Semantic Kernel). Ці інструменти пропонують програмні абстракції для побудови ланцюжків викликів (Chains), ін'єкції контексту (RAG Pipelines) та управління пам'яттю агентів. Однак вони є бібліотеками, а не інструментами управління. Вони не мають візуального інтерфейсу для перегляду та редагування промптів, не забезпечують пошуку по базі шаблонів та не надають механізмів для безпечного управління API-ключами. Фактично, розробник вимушений комбінувати оркестраційний фреймворк із окремим інструментом для управління самими інструкціями, що збільшує складність стеку.

Третьою категорією є інтегровані IDE-плагіни та розширення (наприклад, GitHub Copilot, Continue.dev, Cursor AI). Ці інструменти глибоко вбудовуються у середовище розробки та пропонують генерацію коду та автодоповнення на базі AI.

Проте вони фокусуються на кодогенерації, а не на управлінні контекстом. Розробник не може через Copilot створити, каталогізувати та версіонувати бібліотеку системних інструкцій для різних агентів. Ці інструменти є споживачами промптів, а не менеджерами. Таким чином, виявляється незаповнена ніша - потреба в легковаговому, локальному, клавіатурно-орієнтованому інструменті, який дозволяє повністю контролювати життєвий цикл промпту від створення до використання у мультиагентній системі.

Варто також розглянути категорію настільних додатків на базі фреймворку Electron. Вони надають кросплатформність та звичний веб-інтерфейс, проте їхнім головним недоліком є надмірне споживання системних ресурсів. Electron-додатки по суті запускають повноцінний екземпляр браузера Chromium та середовище Node.js для кожної програми, що призводить до споживання сотень мегабайт оперативної пам'яті навіть у фоновому режимі. Для інструменту, що має працювати паралельно з IDE, базою даних, докер-контейнерами та, можливо, локальними моделями (наприклад, через Ollama), така витрата ресурсів є невиправданою. Це зайвий раз підтверджує доцільність розробки консольного TUI-рішення, яке здатне забезпечити аналогічний функціонал з мінімальними накладними витратами (Overhead).

1.3. Постановка задачі

На основі проведеного глибокого аналізу предметного середовища, виявлених потреб розробників інтелектуальних систем та недоліків існуючих аналогів (як важких SaaS-платформ, так і обмежених CLI-утиліт), було сформульовано розширену постановку задачі. Головною метою кваліфікаційної роботи є створення локальної, високоефективної інформаційної системи (TUI-додатка) для централізованого управління контекстом та інструкціями для AI-моделей. Програмне рішення має функціонувати як єдина точка входу (Single Source of Truth) для управління промптами, забезпечуючи повний цикл підготовки контексту: від його створення та локального зберігання у вигляді структурованих шаблонів до можливості динамічної підстановки параметрів, тестування, керування версіями та валідації результатів перед їхньою відправкою до LLM.

Для досягнення цієї амбітної мети розроблювана система повинна відповідати низці жорстких функціональних вимог. Насамперед, необхідно реалізувати повноцінний та безвідмовний механізм CRUD (Create, Read, Update, Delete) для об'єктів-промптів. Відповідно до цього механізму, система повинна забезпечувати:

– Створення та запис: можливість генерації нових шаблонів інструкцій через інтуїтивний інтерфейс терміналу. Шаблони мають зберігатися локально на файловій системі комп'ютера розробника у зручних для читання та версіонування форматах (переважно Markdown із YAML Frontmatter). Це дозволить зберігати промпти в системі контролю версій Git разом із кодом проєкту, повністю реалізуючи парадигму Prompt-as-Code.

– Читання та вибірка: забезпечення миттєвого доступу до списку всіх наявних шаблонів у проєкті (незалежно від їхньої кількості). Система повинна вміти асинхронно сканувати задані робочі директорії (Workspaces), парсити метадані шаблонів та відображати їхній вміст через оптимізовані механізми роботи з файловою системою (наприклад, через Bun File API для мінімізації затримок I/O). Повинні бути реалізовані швидкі фільтри та алгоритми повнотекстового пошуку.

– Модифікація та оновлення: надання зручного інтерфейсу для редагування існуючих інструкцій безпосередньо у вікні терміналу, з підтримкою системного буфера обміну та автоматичним і атомарним перезаписом змін у файлах.

– Безпечне видалення: можливість швидко та безпечно видаляти застарілі чи непотрібні інструкції з фізичної файлової системи, що автоматично супроводжується синхронізацією кешу бази даних та оновленням візуального стану інтерфейсу.

Крім того, система має задовольняти перелік нефункціональних вимог (Non-Functional Requirements). До них відносяться вимоги щодо продуктивності (Performance): система повинна запускатися менш ніж за 100 мілісекунд та підтримувати стабільні 60 кадрів на секунду (FPS) при відмальовуванні інтерфейсу, щоб забезпечити відчуття плавності. Наступна вимога - це безпека (Security). Оскільки система керує конфігураціями середовища (.env), вона повинна забезпечити ізоляцію API-ключів та токенів авторизації в оперативній пам'яті, унеможливлюючи їхній витік через журнали (Логи).

Вимоги до користувацького інтерфейсу (User Experience / Developer Experience) є не менш важливими. Інтерфейс повинен бути реалізований як реактивний TUI-додаток на базі сучасної бібліотеки React 19. Вибір React для терміналу дозволяє використовувати сучасні парадигми декларативного програмування, де інтерфейс є функцією від поточного стану (UI = f(state)). Для забезпечення високої ергономіки система має підтримувати глибоку обробку подій клавіатури, що дозволить здійснювати швидку та інтуїтивну навігацію по списках, меню та формах без використання комп'ютерної миші. Реактивне відображення стану повинно працювати без ефекту «мерехтіння» (flickering), навіть коли оновлюються великі блоки тексту або здійснюється асинхронне завантаження мережевих даних.

Додатковою функціональною вимогою є підтримка хмарної синхронізації (Cloud Sync). Розробники часто працюють на кількох робочих станціях (офісний комп'ютер, домашній ноутбук, сервер у хмарі). Система повинна забезпечувати можливість двосторонньої синхронізації локальної бази промптів із хмарним сховищем (Google Drive) з автоматичним вирішенням конфліктів версій на основі часових міток (Timestamps). Це гарантуватиме доступність актуальної бази інструкцій незалежно від фізичного розташування розробника.

Нарешті, система повинна підтримувати інтеграцію зі стандартом Model Context Protocol (MCP). Цей відкритий протокол, розроблений компанією Anthropic, визначає стандартизований спосіб комунікації між AI-моделями та зовнішніми інструментами. Підтримка MCP дозволить системі не лише пасивно зберігати промпти, але й активно експортувати їх як інструменти (Tools), які мовна модель може викликати автономно (Function Calling). Це принципово змінює роль системи - від статичного сховища до активного учасника мультиагентної архітектури.

Підсумовуючи, інформаційна система повинна стати центральною та невід'ємною ланкою в екосистемі інструментів AI-розробника, органічно інтегруючись у робочий процес, вирішуючи проблеми фрагментації контексту та значно прискорюючи цикл експериментів з великими мовними моделями.

Важливою функціональною вимогою також є забезпечення безперешкодної інтеграції системи з існуючими конвеєрами безперервної інтеграції та розгортання (CI/CD). В індустріальних проєктах промпти часто потребують автоматизованого тестування (наприклад, на наборах даних Golden Datasets) перед деплоєм у виробниче середовище. Оскільки розроблювана інформаційна система базується на локальному зберіганні файлів у форматі Markdown, вона природно підтримує такі CI/CD процеси. Кожна зміна промпту відслідковується у системі контролю версій Git, що дозволяє тригерити пайплайни тестування, обчислювати метрики якості згенерованих відповідей (наприклад, BLEU, ROUGE або семантичну подібність) та забезпечувати високу надійність поведінки AI-агентів після кожного оновлення інструкцій.

Висновки до розділу 1

У першому розділі було детально та всебічно проаналізовано предметне середовище розробки систем штучного інтелекту. Висвітлено історичну еволюцію підходів до інженерії підказок (Prompt Engineering) - від базових текстових запитів до складних структур типу Chain-of-Thought. Виявлено, що жорстке кодування інструкцій у коді (Hardcoded-пастка), відсутність системи версіонування та фрагментація логіки побудови контексту призводять до феномену дрейфу інструкцій (Prompt Drift), вразливостей контекстного вікна та значного зниження продуктивності розробників при масштабуванні корпоративних проєктів.

Детальний аналіз існуючих ринкових рішень показав, що сучасні «важкі» SaaS-платформи, незважаючи на потужний аналітичний функціонал, створюють додаткове тертя (friction) у процесі розробки через необхідність постійного контекстного перемикання між браузером та інтегрованим середовищем розробки (IDE). Більше того, вони ускладнюють локальне тестування AI-агентів (Local First Development) та несуть ризики компрометації конфіденційних даних. Для вирішення цих проблем обґрунтовано необхідність застосування концепції Prompt-as-Code та використання термінального інтерфейсу (TUI) як ідеального компромісу між швидкістю інструментів командного рядка (CLI) та наочністю і зручністю

повноцінного графічного інтерфейсу (GUI), що дозволяє керувати промптами повністю за допомогою клавіатури.

На завершення розділу було сформульовано розгорнуту постановку задачі на розробку інформаційної системи, що включає як функціональні, так і нефункціональні вимоги. Визначено, що система повинна гарантувати повний цикл CRUD-операцій для шаблонів інструкцій, забезпечувати їх надійне локальне зберігання у форматі Markdown із YAML Frontmatter, гарантувати високу продуктивність (60 FPS) та безпеку конфіденційних даних. Розроблена система має надавати зручний, реактивний термінальний інтерфейс, що дозволить розробникам ефективно інтегрувати управління контекстом у свій щоденний робочий процес без шкоди для продуктивності та безпеки.

РОЗДІЛ 2. ІНФОРМАЦІЙНЕ ТА МАТЕМАТИЧНЕ ЗАБЕЗПЕЧЕННЯ

2.1. Аналіз предметної області (вхідні та вихідні дані)

Інформаційна система функціонує в середовищі, де головним об'єктом обробки є неструктурований та напівструктурований текст. Для забезпечення надійної взаємодії системи з файловим сховищем, інструментами редагування та кінцевим споживачем інформації (API великих мовних моделей), необхідно чітко формалізувати моделі вхідних та вихідних даних. Вхідними даними системи виступають статичні текстові файли шаблонів, які зберігаються на фізичному накопичувачі, системні змінні оточення та динамічний користувацький ввід, що надходить через інтерфейс терміналу.

Основним форматом зберігання інструкцій обрано стандарт Markdown (MD), що є де-факто стандартом розмітки у відкритому програмному забезпеченні (Open Source). Markdown дозволяє комбінувати читабельний для людини текст із форматуванням (заголовки, списки, блоки коду), що ідеально підходить для написання багаторядкових системних підказок. З математичної точки зору, синтаксис Markdown можна представити як контекстно-вільну граматику, де кожний елемент форматування є токеном, який парсер може перетворити на абстрактне синтаксичне дерево (AST).

Більш формально, AST можна визначити як орієнтований ациклічний граф $G = (V, E)$, де V - множина вершин (вузлів синтаксичного дерева), а E - множина дуг, що відображають ієрархічні відношення. Кожна вершина $v \in V$ характеризується кортежем (T, C, A), де T - тип вузла (наприклад, Heading, Paragraph, List), C - множина дочірніх вузлів, а A - асоціативний масив атрибутів (наприклад, рівень заголовка). Таке математичне подання є критично важливим для реалізації алгоритмів валідації контексту, оскільки дозволяє рекурсивно обходити дерево (наприклад, алгоритмом пошуку в глибину - DFS) та програмно вилучати специфічні блоки тексту, призначені для конкретних AI-агентів, ігноруючи непотрібну розмітку.

Ключовою інновацією в управлінні інформаційним забезпеченням є використання специфікації YAML Frontmatter. YAML Frontmatter - це блок

структурованих метаданих, що розміщується на початку Markdown-файлу і відділяється трьома дефісами (---). Цей блок використовується для зберігання декларативних параметрів промпту, які не є частиною безпосередньої текстової інструкції, але необхідні для її коректного виконання. Структура вхідних метаданих (JSON Schema) може бути описана наступним чином:

- id (String): глобальний унікальний ідентифікатор промпту типу UUIDv4, необхідний для гарантування унікальності навіть при зміні назви файлу.

- title (String): людиночитабельна назва інструкції, яка відображається в інтерфейсі (наприклад, 'Генератор Unit-тестів').

- model (String): пріоритетна мовна модель, для якої оптимізовано промпт (наприклад, 'gpt-4o' або 'claude-3-opus-20240229').

- temperature (Float): параметр гіперконфігурації в діапазоні [0.0; 2.0], який визначає ступінь креативності моделі. Значення наближені до 0.0 використовуються для детермінованих завдань (наприклад, написання коду), тоді як значення > 1.0 використовуються для творчої генерації тексту.

- tags (Array<String>): масив текстових тегів для швидкої категоризації та пошуку в базі даних (наприклад, ['frontend', 'react', 'debugging']).

- variables (Array<Object>): масив об'єктів, що описують необхідні змінні контексту. Кожна змінна має ім'я, тип даних та опис. Ці змінні пізніше замінюються на реальні дані за допомогою синтаксису шаблонізації (наприклад, {{user_code}}).

Вихідними даними системи є форматовані, зібрані контексти, які генеруються після підстановки всіх необхідних змінних та секретних ключів. Формат цих даних строго регламентується специфікаціями API провайдерів мовних моделей (OpenAI REST API або Anthropic Messages API). Зазвичай це складний JSON-об'єкт типу Payload, який містить масив повідомлень 'messages'. Кожне повідомлення має атрибут 'role' (system, user, assistant) та 'content'. Завдання інформаційної системи - гарантувати, що локальні Markdown-шаблони будуть безпомилково перетворені на цей кінцевий JSON-формат, зберігши недоторканними всі логічні блоки та теги форматування.

Окремо варто зазначити роль конфігураційних файлів '.env' як критичного інформаційного ресурсу. Ці файли містять секретні дані - API-ключі, токени авторизації, URL-адреси ендпоінтів та інші параметри, необхідні для взаємодії з зовнішніми сервісами. Система повинна забезпечити безпечне зчитування цих файлів під час ініціалізації, їхнє зберігання виключно в оперативній пам'яті процесу (без серіалізації на диск) та безпечну ін'єкцію значень у HTTP-заголовки при формуванні запитів до LLM API. Будь-який витік цих даних через логи, кеш або тимчасові файли є неприпустимим з точки зору інформаційної безпеки.

Динамічний користувацький ввід є третім типом вхідних даних. Він надходить через інтерфейс терміналу у вигляді послідовностей натискань клавіш (Keystrokes), які перехоплюються ядром OpenTUI та перетворюються на React-події. Цей ввід включає текстові дані (назви файлів, пошукові запити), навігаційні команди (стрілки, Tab, Enter, Escape) та системні комбінації (Ctrl+C для завершення, Ctrl+V для вставки з буфера обміну). Коректна обробка цього потоку даних вимагає побудови складного конвеєра парсингу, який буде описаний у підрозділі 2.3.

Для глибшого математичного аналізу підготовки даних (Retrieval-Augmented Generation, RAG) варто розглянути теоретико-інформаційні основи стиснення контексту. Системи RAG є життєво необхідними, оскільки неможливо вкласти всі корпоративні знання у жорсткий ліміт токенів моделі (Context Window). Завданням RAG є вибірка найрелевантніших фрагментів тексту з великого корпусу документів. Математично цей процес спирається на векторні представлення (Embeddings). Текстовий документ перетворюється за допомогою спеціалізованої моделі (наприклад, text-embedding-ada-002) у N-вимірний вектор $V \in R^n$ (де n зазвичай дорівнює 1536 або 3072).

Обчислення семантичної подібності між запитом користувача (Query Vector - Q) та базою знань (Document Vector - D) базується на розрахунку косинусної подібності (Cosine Similarity). Косинусна подібність обчислюється як скалярний добуток векторів, поділений на добуток їхніх евклідових норм. Ця метрика нормалізована у діапазоні [-1, 1], де 1 означає абсолютну семантичну тотожність, 0 - ортогональність (відсутність зв'язку), а -1 - протилежний зміст. Для швидкого

пошуку найближчих векторів (k-Nearest Neighbors) у просторі високої розмірності використовуються спеціалізовані Векторні Бази Даних (наприклад, ChromaDB, Pinecone, Qdrant), які імплементують алгоритми наближеного пошуку (Approximate Nearest Neighbors - ANN), такі як HNSW (Hierarchical Navigable Small World).

Додатково, процес фільтрації контексту можна аналізувати через призму теорії інформації Шеннона. Зайві або дубльовані інструкції у промпті створюють «інформаційний шум», який зменшує ентропію корисного сигналу, ускладнюючи моделі видобування релевантної інформації (явище Lost in the Middle). Інформаційна система управління промптами повинна мінімізувати цю ентропію шляхом модульної збірки: замість одного монолітного файлу, інструкції розбиваються на атомарні компоненти (наприклад, System Role, Constraints, Output Format), які динамічно конкатенуються залежно від конкретного запиту користувача.

2.2. Проектування архітектури системи

Архітектура розроблюваного проєкту є унікальною і суттєво відрізняється від класичних патернів побудови веб- або десктопних застосунків. Це зумовлено використанням інноваційного стеку технологій: середовища виконання Bun, бібліотеки React 19 та спеціалізованого консольного фреймворку OpenTUI. Цей стек дозволяє перенести парадигму декларативного створення інтерфейсів, яка домінує у браузерах, безпосередньо у середовище текстового терміналу операційної системи.

Ядром візуальної архітектури є відмова від браузерного DOM (Document Object Model). Замість традиційних HTML-тегів, таких як , або <input>, система використовує кастомні абстракції OpenTUI: <box>, <text>, <scrollbox>. Ці компоненти не підтримують CSS. Натомість вони використовують власну алгебраїчну систему стилізації, що імітує поведінку Flexbox. Це створює закриту, високоефективну екосистему компонентів, яка абсолютно не залежить від важких браузерних рушіїв Blink чи WebKit. Відображення інтерфейсу у терміналі здійснюється у вигляді послідовності байтів, що передаються у стандартний потік виводу (stdout), що забезпечує мікросекундні затримки при рендерингу та дозволяє додатку працювати в рази швидше за аналоги на базі фреймворку Electron.

Глобальна архітектура системи розбита на незалежні, слабко зв'язані модулі за принципом Atomic Modular Architecture (атомарна модульна архітектура). Проєкт організовано у директорії 'src/modules', яка містить підсистеми: 'files' (основний файловий менеджер), 'env-settings' (для управління конфігураціями та змінними оточення), 'gdrive' (хмарна синхронізація з Google Drive) та 'mcp' (Model Context Protocol). Інші експериментальні підходи до побудови моноліту були відкинуті на етапі раннього тестування. Такий атомарний модульний підхід гарантує, що критична помилка в одному ізольованому модулі не призведе до каскадного падіння всього термінального інтерфейсу.

Для взаємодії між цими незалежними модулями імплементовано архітектурний патерн Mediator (Посередник). У традиційному React-застосунку компоненти часто змушені передавати властивості (props) на багато рівнів вниз по ієрархії дерев компонентів (явище Prop Drilling) або використовувати важкий React Context, який провокує непотрібні перемальовування. У розробленій системі Mediator виступає як центральна шина подій (Event Bus). Модулі не викликають один одного напряму; вони публікують формалізовані події та запити (наприклад, 'FILE_SELECTED' або 'DELETE_FILE_COMMAND') та підписуються на події. Це архітектурно знижує зачеплення (Coupling) коду та значно полегшує його підтримку та модульне тестування (Unit Testing).

Особливістю імплементації Mediator у даному проєкті є застосування принципів CQRS-lite (Command Query Responsibility Segregation). Усі операції розділені на дві незалежні групи: команди (Commands), які мутують стан системи (наприклад, запис нового промпту на диск), та запити (Queries), які лише зчитують дані без їхньої зміни (наприклад, пошук файлів). Математично цей поділ гарантує, що функції-запити є чистими функціями (Pure Functions) без побічних ефектів, що дозволяє застосовувати агресивне кешування результатів та мемоїзацію на рівні UI-компонентів, уникаючи дорогих звернень до бази даних.

Управління глобальним станом (Global State Management) реалізовано за допомогою бібліотеки Zustand. Вибір Zustand замість традиційного інструменту Redux продиктований жорсткими вимогами до продуктивності в умовах терміналу.

Zustand не потребує обгортання додатка у Context Provider, що зменшує глибину абстрактного дерева компонентів та пришвидшує етап рендерингу. Крім того, архітектура Zustand дозволяє отримувати доступ до глобального стану поза межами React-компонентів, що є архітектурно необхідним для обробки низькорівневих системних сигналів терміналу (наприклад, обробка сигналу SIGINT для безпечного закриття додатка та збереження бази даних перед виходом).

Архітектура введення (Input Handling Architecture) також є глибоко специфічною. У вебі розробник просто підписується на стандартну подію 'onChange' елемента <input>. В OpenTUI немає браузерного циклу подій (Event Loop) та обробників подій DOM. Натомість, ядро фреймворку перехоплює сирий потік 'stdin' операційної системи через API середовища Bun, побайтово парсить складні ANSI-послідовності, які генеруються терміналом при натисканні клавіш, та штучно синтезує React-події (Synthetic Events). Для текстових полів введення система повинна самостійно вираховувати математичні координати (X та Y) положення миготливого курсора на матриці екрану, обробляти складні комбінації клавіш (наприклад, Ctrl+V для вставки інформації з буфера обміну) та розраховувати зсув тексту при виході символів за межі поля (Horizontal Scrolling Algorithm).

Підсистема зберігання даних реалізує гібридну плоску архітектуру зберігання інформації. Фізичні дані (Markdown-шаблони) лежать безпосередньо на диску для забезпечення легкості їх редагування у зовнішніх редакторах, таких як VS Code. Водночас для швидкого повнотекстового пошуку та миттєвого відображення списків система використовує локальну базу даних SQLite. Взаємодія з базою абстрагована за допомогою сучасного інструменту Drizzle ORM (Object-Relational Mapping). Drizzle ORM проектує реляційні таблиці SQL у строгу типізацію на рівні TypeScript (AST), дозволяючи виявляти помилки у типах чи назвах колонок ще на етапі компіляції коду. Це гарантує максимальну стабільність міграцій бази даних та пришвидшує розробку системи кешування.

Важливим аспектом архітектури є система кастомних React-компонентів для терміналу. У проєкті реалізовано ієрархію компонентів: від низькорівневих примітивів (FocusableBox, ScrollableList) до високорівневих композицій

(FileExplorer, SettingsEditor, McpToolsList). Кожен компонент є автономною одиницею, яка інкапсулює власну логіку обробки клавіатурних подій, відмальовування та управління фокусом. Такий підхід дозволяє створювати складні інтерфейси шляхом композиції простих елементів, що є фундаментальним принципом React-парадигми.

Окрему увагу заслуговує проектування системи навігації між модулями. Оскільки у терміналі відсутній концепт URL-адрес, маршрутизація реалізована через стек екранів (View Stack). Кожен модуль реєструє свої екрани у центральному роутері. При переході між модулями роутер виконує демонтаж (unmount) поточного React-дерева компонентів та монтаж (mount) нового. Це звільняє підписки на клавіатурні події та таймери, запобігаючи витокам пам'яті (Memory Leaks) та конфліктам обробників клавіш між різними модулями.

2.3. Математичне та алгоритмічне забезпечення

Відмова від готових браузерних рушіїв на користь малювання інтерфейсу безпосередньо у терміналі вимагає розробки значної кількості математичного та алгоритмічного забезпечення. У терміналі екран являє собою двовимірну матрицю символів (сітку) M(W, H), де W - кількість колонок екрану, а H - кількість рядків. Кожна клітинка цієї матриці M[x, y] містить не лише сам символ utf-8, але й метадані його стилізації: колір тексту (foreground color), колір фону (background color), та модифікатори стилю (жирний, курсив, підкреслений).

Першою фундаментальною алгоритмічною проблемою є реконсиляція віртуального графа інтерфейсу (Reconciliation). Оскільки малювання у терміналі через стандартний потік виводу (stdout) є дуже дорогою операцією, повне очищення екрану та перемальовування всієї матриці M[x, y] при кожній зміні стану компонента (наприклад, при блиманні курсора) призведе до сильного візуального 'мерехтіння' (flickering) та високого навантаження на процесор. Для вирішення цієї задачі використовується модифікований евристичний алгоритм диференціального обчислення дерев, подібний до того, що застосовується у React Fiber. Алгоритм порівнює поточний граф віртуальних вузлів (Current Virtual Node Graph) із новим

графом, що утворився після зміни стану (Next Virtual Node Graph). Обчислювальна складність наївного порівняння двох дерев становить O(n^3), що неприпустимо для продуктивності 60 FPS. Застосування евристик на основі унікальних ключів (Keys) компонентів знижує алгоритмічну складність до O(n). Результатом порівняння є мінімальний набір патчів (Diff Patches) - точних координат (x, y), які дійсно змінили свій вміст і потребують оновлення в матриці терміналу.

Другою математичною задачею є вирішення системи лінійних рівнянь розміщення компонентів (Layout Engine Math). У вебі цим займається браузерний двигун розмітки (CSS Layout Engine). В OpenTUI реалізовано власну систему алгоритмів розміщення на основі Flexbox. Для обчислення координат кожного вузла система рекурсивно проходить граф компонентів згори донизу. Розмір батьківського контейнера (Width, Height) розподіляється між дочірніми елементами згідно з їхніми коефіцієнтами 'flexGrow', 'flexShrink' та фіксованими 'width'. Математична формула обчислення ширини гнучкого елемента wi обчислюється як базова ширина (flexBasis) плюс пропорційна частка від вільного простору (Available Space), поділена на суму всіх коефіцієнтів flexGrow. Якщо вільний простір є від'ємним, система застосовує рівняння пропорційного звуження на основі 'flexShrink'. Ця алгебраїчна система гарантує, що інтерфейс буде коректно адаптуватися при зміні фізичних розмірів вікна терміналу користувачем (Resize Event).

З погляду лінійної алгебри, рендеринг у терміналі можна описати як операцію трансформації над двовимірними матрицями. Вікно терміналу - це матриця T розмірності WxH, де елемент t_ij представляє піксельний блок символу. Кожен візуальний компонент TUI (наприклад, вікно списку файлів або діалогове вікно) спочатку відмальовується у власну локальну матрицю L (Off-screen Canvas) за допомогою афінних перетворень для обчислення абсолютних координат. Фінальний рендеринг вимагає операції накладання (Compositing), де значення матриці L записуються поверх матриці T з урахуванням z-індексу (глибини). Для оптимізації цього процесу застосовується диференціальне порівняння матриць (Matrix Diffing): замість повного перезапису матриці T, система обчислює різницю $\Delta T = T_{new} - T_{old}$. Лише ті елементи матриці, де $\Delta T_{ij} \neq 0$, серіалізуються у вигляді керувальних

ANSI-кодів та відправляються у потік stdout, що дозволяє досягти теоретичної межі пропускної здатності I/O.

Третім алгоритмічним викликом є реалізація механізму віртуальної прокрутки (Virtual Scrolling). При наявності тисяч промптів у базі даних (наприклад, під час відображення великого списку у модулі 'files'), створення тисяч об'єктів <box> та <text> у пам'яті призведе до її переповнення та катастрофічного падіння продуктивності рендерера. Алгоритм віртуалізації списків розв'язує цю проблему шляхом відмальовування лише тієї частини елементів (Вікна перегляду - Viewport), яка фактично видима на екрані матриці H. Алгоритм розраховує математичний зсув (Scroll Offset - Y_offset), який залежить від розміру кожного елемента списку (Item Height). Кількість видимих елементів розраховується за формулою N = Viewport Height / Item Height. Система динамічно підміняє дані у цих N компонентах при отриманні подій від клавіш 'Стрілка вгору' чи 'Стрілка вниз', забезпечуючи константне споживання пам'яті O(1) незалежно від обсягу бази даних SQLite.

Четвертим важливим алгоритмічним елементом є парсинг багатобайтових клавіатурних подій. Коли користувач натискає складну клавішу (наприклад, стрілку або комбінацію Alt+F), термінал не відправляє один байт. Він відправляє ANSI Escape Sequence - послідовність байтів, яка починається з символу ESC (ASCII 27). Наприклад, натискання стрілки вгору генерує послідовність '[A'. Для розпізнавання цих послідовностей у реальному часі зі швидкого потоку 'stdin', система використовує алгоритмічну модель Скінченного Автомата (Finite State Machine, FSM). Автомат має різні стани переходу (Start State, Escape State, Control State). Якщо приходить байт '27', автомат переходить у стан очікування наступного байта. Ця математична модель дозволяє безпомилково розпізнавати комбінації клавіш, ігноруючи 'сміття', і делегувати ці натискання компонентам React через систему синтетичних подій (Synthetic Events).

Математична модель скінченного автомата може бути формалізована як кортеж (Σ, S, s0, δ, F), де Σ - вхідний алфавіт (всі можливі байти потоку stdin), S - скінченна множина станів парсера, s0 - початковий стан (очікування введення), δ - функція переходів, яка детерміновано визначає наступний стан на основі поточного

стану та отриманого байта, а F - множина кінцевих станів, що відповідають розпізнаній валідній комбінації клавіш. Часова складність обробки одного символу таким автоматом є O(1), що забезпечує миттєву реакцію інтерфейсу на дії користувача без використання регулярних виразів (RegEx), які могли б призвести до катастрофічного падіння продуктивності (ReDoS).

Нарешті, математичного обґрунтування вимагає робота з реляційною алгеброю бази даних. Модуль 'files' використовує Drizzle ORM для трансформації об'єктно-орієнтованих запитів (TypeScript) у математичні операції реляційної алгебри: проекції (SELECT), селекції (WHERE) та декартові добутки/з'єднання (JOIN). Використання B-Tree індексів у драйвері SQLite дозволяє зменшити обчислювальну складність пошуку конкретного шаблону промпту за його тегом чи назвою з лінійного часу O(n) до логарифмічного O(log n). Алгоритми хешування (SHA-256) використовуються для обчислення контрольних сум файлів, що гарантує швидку інвалідацію кешу без необхідності порядкового порівняння текстового вмісту великих Markdown-документів.

Висновки до розділу 2

У другому розділі проведено надзвичайно детальний та глибокий аналіз інформаційного та математичного забезпечення розробленої системи. Визначено критичну роль статичних шаблонів Markdown із метаданими YAML Frontmatter, динамічних контекстних змінних та конфігураційних файлів для формування фінальних системних промптів у вигляді структурованих JSON-даних (Payload) для API великих мовних моделей.

Детально розкрито унікальну архітектуру проєкту, побудовану на базі принципів Atomic Modular Architecture та патерну CQRS-lite. Обґрунтовано відмову від традиційного браузерного DOM на користь кастомних термінальних абстракцій (<box>, <text>), що забезпечило наднизькі затримки. Описано використання патерну Mediator для декуплінгу робочих модулів (files, env-settings, gdrive, mcp), застосування Drizzle ORM для безпечної та строго типізованої роботи з локальною

базою даних SQLite, а також використання бібліотеки Zustand для управління глобальним станом (Global State) без проблеми Prop Drilling.

Широко описано математичне та алгоритмічне підґрунтя функціонування системи. Математично обґрунтовано застосування евристичного алгоритму реконсиляції (Reconciliation) віртуального графа інтерфейсу, що знижує обчислювальну складність оновлення екрану з кубічної O(n^3) до лінійної O(n). Формалізовано складні математичні моделі розв'язання систем лінійних рівнянь розміщення (Flexbox Layouting) у дискретній матриці терміналу. Детально розглянуто алгоритми віртуальної прокрутки (Virtual Scrolling) для забезпечення константного споживання пам'яті O(1) та використання алгоритмічних моделей скінченних автоматів (FSM) для ефективного парсингу керувальних ANSI-послідовностей клавіатурних подій.

РОЗДІЛ 3. ПРОГРАМНЕ ТА ТЕХНІЧНЕ ЗАБЕЗПЕЧЕННЯ

3.1. Засоби розробки

Розробка сучасної інформаційної системи вимагає ретельного підбору інструментарію, який забезпечить не лише високу швидкість виконання, але й зручність підтримки кодової бази, її масштабованість та безпеку. Для реалізації системи централізованого управління контекстом AI-систем було обрано стек технологій, що базується на середовищі Bun, мові TypeScript, бібліотеці React 19 та спеціалізованому фреймворку OpenTUI. Кожен із цих інструментів відіграє критичну роль у забезпеченні загальної працездатності комплексу.

Основною мовою програмування обрано TypeScript. Це строго типізована надбудова над JavaScript, розроблена компанією Microsoft. Використання TypeScript дозволяє виявляти абсолютну більшість архітектурних та логічних помилок ще на етапі компіляції (Compile-Time), до фактичного запуску коду. Статична типізація є критично важливою для розробки великих систем, де взаємодіють десятки модулів. Вона забезпечує надійність рефакторингу, покращує автодоповнення в інтегрованих середовищах розробки (IDE) та слугує самодокументованим кодом. Зокрема, у проєкті інтенсивно використовуються утилітарні типи (Utility Types), дженерики (Generics) та строгі інтерфейси (Interfaces) для опису контрактів між компонентами Mediator та обробниками подій.

Середовищем виконання (Runtime Environment) виступає Bun. Історично більшість серверних JavaScript-застосунків створювалися на базі Node.js. Проте Node.js, маючи багату екосистему, страждає від повільного часу запуску (Cold Start) та високого споживання пам'яті через застарілі механізми розділення модулів (CommonJS). Іншою альтернативою міг би стати Deno, який пропонує кращу безпеку 'з коробки', проте він також використовує рушій V8 від Google, який схильний до надмірного резервування оперативної пам'яті. Bun - це принципово нове середовище, написане з нуля на низькорівневій мові Zig та побудоване навколо рушія JavaScriptCore (який використовується у браузері Safari). JavaScriptCore ідеально підходить для короткоживучих процесів, якими часто є утиліти командного

рядка (CLI). Використання Bun дозволило скоротити час запуску TUI-додатка до мілісекунд, що є критично важливим для забезпечення безперервного потоку роботи інженера (Developer Flow). Крім того, Bun надає високоефективні нативні API для роботи з файловою системою (Bun.file()), які використовують механізми операційної системи для мінімізації накладних витрат при читанні та записі файлів.

Для побудови візуального інтерфейсу було обрано React 19. Хоча React традиційно сприймається виключно як інструмент для розробки веб-сторінок, його архітектура (React Fiber) дозволяє відокремити логіку обчислення стану та життєвих циклів (Reconciler) від безпосереднього відмальовування (Renderer). React 19 привносить низку інновацій, таких як покращене управління побічними ефектами, автоматичне батчування оновлень стану (Automatic Batching) та оптимізований механізм конкурентного рендерингу (Concurrent Mode). Конкурентний режим особливо важливий для терміналу: він дозволяє переривати важкі завдання відмальовування (наприклад, форматування великого Markdown-файлу на 5000 рядків) для негайної обробки високопріоритетних подій, таких як натискання клавіші користувачем. Завдяки хукам типу useTransition інтерфейс залишається завжди чуйним (Responsive).

Відмальовування компонентів React у терміналі здійснюється за допомогою спеціалізованого фреймворку OpenTUI (пакети @opentui/react та @opentui/core). Цей фреймворк виступає в ролі кастомного рендерера для React. Він замінює браузерний DOM на власну абстракцію, що працює виключно з текстовими символами та керувальними ANSI-послідовностями. OpenTUI надає набір примітивів: <box> для створення контейнерів, <text> для виведення тексту, та системи розрахунку компонування, яка імітує поведінку CSS Flexbox. Це усуває необхідність використання важких графічних рушіїв, роблячи додаток надзвичайно швидким та ресурсоефективним.

Для управління даними використовується Drizzle ORM у поєднанні з вбудованим драйвером bun:sqlite. Drizzle ORM - це сучасний TypeScript-first інструмент об'єктно-реляційного відображення. На відміну від традиційних ORM (таких як TypeORM або Prisma), Drizzle пропонує максимальну близькість до сирого

SQL, водночас забезпечуючи строгу типізацію на рівні схеми бази даних. Це означає, що розробник може писати складні запити з автодоповненням та перевіркою типів, не втрачаючи продуктивності. Важливою перевагою Drizzle є генерація математично точних міграцій (Migrations). Кожна зміна в моделях (наприклад, додавання колонки 'tags') компілюється в атомарний SQL-файл, який містить як команди накату (Up), так і відкату (Down). Це гарантує цілісність даних при оновленні версій. SQLite, зі свого боку, є найпоширенішою у світі вбудованою реляційною базою даних. Вона не вимагає встановлення окремого сервера, зберігає всі дані у локальному файлі та забезпечує транзакційність (ACID) операцій. Використання SQLite ідеально підходить для архітектури настільних утиліт, де потрібно забезпечити швидкий локальний пошук та індексування без мережевих затримок.

Нарешті, для управління глобальним станом додатку (Global State Management) було обрано бібліотеку Zustand. Традиційні рішення, такі як Redux, вимагають написання великої кількості шаблонного коду (Boilerplate), створення редьюсерів, екшенів та типів. Zustand пропонує мінімалістичний підхід, створюючи глобальний стор (Store) на основі хуків, який можна легко підключати до будь-якого компонента. Більше того, Zustand дозволяє читати та оновлювати стан поза контекстом React (наприклад, у фонових воркерах або обробниках системних переривань), що є архітектурно необхідним для термінального середовища.

3.2. Вимоги до технічного та програмного забезпечення

Оскільки розроблена інформаційна система функціонує як TUI-застосунок, вимоги до апаратного та програмного забезпечення суттєво відрізняються від традиційних веб- чи десктопних додатків на базі Electron. Головною перевагою консольної архітектури є її надзвичайна легковаговість та низькі вимоги до обчислювальних ресурсів, що дозволяє запускати систему навіть на застарілому або малопотужному обладнанні.

Вимоги до апаратного забезпечення (Hardware Requirements):

1. Центральний процесор (CPU): Архітектура x86-64 (Intel Core i3/i5/i7, AMD Ryzen) або ARM64 (Apple Silicon M1/M2/M3, процесори сімейства Snapdragon для

Windows). Завдяки компіляції Bun за допомогою LLVM та оптимізації рушія JavaScriptCore, система не вимагає високої багатопотоковості. Мінімальна тактова частота - від 1.0 ГГц.

2. Оперативна пам'ять (RAM): Мінімальний обсяг пам'яті для роботи системи складає 128 МБ. Рекомендований обсяг - 512 МБ (для роботи з великими базами шаблонів, що налічують тисячі файлів, та індексування їх у SQLite). Відсутність браузерного рушія (який зазвичай споживає від 500 МБ до кількох ГБ пам'яті) дозволяє системі працювати надзвичайно ефективно.

3. Постійний накопичувач (Storage): Система потребує близько 100 МБ вільного простору на жорсткому диску (HDD) або твердотільному накопичувачі (SSD) для зберігання середовища Bun, залежностей проєкту (node_modules) та локальної бази даних кешу. Для забезпечення максимальної швидкості ледачого завантаження файлів (Lazy Loading) настійно рекомендується використання NVMe SSD.

4. Відеоадаптер (GPU): Система не використовує апаратне прискорення графіки (Hardware Acceleration). Робота здійснюється виключно через програмний рендеринг символів у терміналі, тому наявність спеціалізованої дискретної відеокарти не вимагається.

Вимоги до програмного забезпечення (Software Requirements):

1. Операційна система (OS): Проєкт є кросплатформним та підтримує всі сучасні операційні системи: Windows 10/11 (через Windows Subsystem for Linux - WSL 2 або нативно через PowerShell), macOS (починаючи з версії 10.15 Catalina) та більшість дистрибутивів Linux (Ubuntu, Debian, Fedora, Arch Linux).

2. Емулятор терміналу (Terminal Emulator): Для коректного відображення TUI-інтерфейсу, роботи системи кольорів (True Color 24-bit) та правильного парсингу клавіатурних комбінацій потрібен сучасний емулятор терміналу. Для користувачів Windows рекомендується використання Windows Terminal (версія 1.15 або новіша). Для користувачів macOS - iTerm2, Alacritty або Kitty. Стандартний 'Command Prompt' (cmd.exe) у старих версіях Windows не підтримується через відсутність повної підтримки ANSI-послідовностей.

3. Мережеве з'єднання та безпека: Для локальної генерації промптів підключення до Інтернету не потрібне, проте для роботи з LLM API вимагається захищене з'єднання з підтримкою протоколу TLS 1.3. Оскільки система передає конфіденційні промпти через мережу, обмін даними обов'язково повинен бути зашифрований, що гарантує захист від атак 'Man-in-the-Middle' (MitM). У корпоративних мережах (де може застосовуватись mTLS - взаємна автентифікація) система дозволяє налаштовувати кастомні сертифікати за допомогою глобальних параметрів Bun.

3.3. Опис програмної реалізації

Програмна реалізація системи базується на строгих принципах програмної інженерії, зокрема на застосуванні шаблонів проєктування (Design Patterns), модульності (Modularity) та слабкого зв'язування (Loose Coupling). Вся кодова база написана мовою TypeScript з дотриманням принципів SOLID та Clean Architecture. Коренева структура проєкту логічно поділяється на ядро системи ('src/core') та бізнес-модулі ('src/modules').

Ядро системи відповідає за інфраструктурні завдання: управління пам'яттю, маршрутизацію, управління залежностями та міжмодульну взаємодію. Найважливішим архітектурним рішенням у ядрі є впровадження кастомного контейнера інверсії управління (Dependency Injection Container), розташованого у директорії 'src/core/di'.

Впровадження залежностей (DI) - це патерн, який дозволяє класу не створювати свої залежності самостійно (через оператор new), а отримувати їх ззовні. Це кардинально спрощує модульне тестування (Unit Testing), оскільки реальні сервіси можна легко замінити на мок-об'єкти (Mock Objects). Розроблений DI-контейнер є унікальним для даного проєкту і підтримує управління життєвими циклами об'єктів. Життєвий цикл Singleton гарантує, що у системі існує лише один екземпляр класу. Це використовується для таких критичних компонентів, як пул з'єднань з базою даних SQLite (Database Connection Pool) або глобальний логер (Logger). Життєвий цикл Transient гарантує створення нового екземпляра об'єкта при

кожному зверненні до контейнера, що є корисним для тимчасових сервісів валідації або розрахунків. Життєвий цикл Scoped створює один екземпляр у межах певного контексту (наприклад, життєвого циклу одного модального вікна).

Для забезпечення ізоляції між графічним інтерфейсом користувача (UI) та бізнес-логікою було реалізовано патерн Mediator, який концептуально наслідує підхід CQRS-lite (Command Query Responsibility Segregation). Релізація медіатора розташована в 'src/core/mediator'.

У класичних React-застосунках бізнес-логіка часто змішується з логікою рендерингу безпосередньо всередині компонентів, що призводить до створення так званих 'God Components'. Патерн Mediator повністю вирішує цю проблему, виступаючи як центральна шина подій (Event Bus). Взаємодія будується навколо двох типів об'єктів: Запитів (Requests) та Обробників (Handlers). Запит - це простий клас даних (Data Transfer Object), який описує намір користувача, наприклад, 'DeletePromptCommand' або 'FetchPromptsQuery'. Обробник - це клас, що імплементує інтерфейс 'IRequestHandler', і містить безпосередню бізнес-логіку виконання запиту.

Коли користувач натискає клавішу 'Delete' в інтерфейсі, React-компонент не звертається до файлової системи чи бази даних напряму. Він створює об'єкт 'DeletePromptCommand' і передає його методу 'mediator.send()'. Mediator, використовуючи DI-контейнер, знаходить відповідний 'DeletePromptHandler', ініціалізує його, впроваджує необхідні залежності (наприклад, FileSystemRepository) і викликає метод 'handle()'. Цей підхід робить код інтерфейсу максимально 'тупим' (Dumb Components), залишаючи його відповідальним лише за відображення даних та перехоплення клавіш.

Третім стовпом інфраструктурного ядра є динамічний проксі-маршрутизатор (Router), розміщений у 'src/core/router'. У вебі маршрутизація базується на зміні URL-адреси. У терміналі цього концепту не існує. Розроблений роутер керує стеком активних екранів (View Stack). Коли користувач переходить у нове меню, роутер монтує відповідний React-компонент, одночасно демонтуючи попередній. Це

гарантує звільнення оперативної пам'яті та припинення прослуховування подій клавіатури неактивними екранами, що запобігає витокам пам'яті (Memory Leaks).

Бізнес-логіка системи поділена на незалежні предметні області (Домени), що відповідає парадигмі Domain-Driven Design (DDD). Усі модулі розташовані в директорії 'src/modules'. Основною перевагою такої архітектури є те, що модулі слабко зв'язані між собою. Кожен модуль інкапсулює свою базу даних (через ізольовані схеми Drizzle), свої обробники Mediator та свої React-компоненти.

Для ефективної роботи з ієрархічними інтерфейсами у системі була спроєктована архітектура проксі-маршрутизатора (Proxy Router). На відміну від веб-додатків, які покладаються на History API та URL-шляхи (наприклад, '/prompts/edit/123'), TUI-додаток використовує концепцію Стек Екранів (View Stack). Це структура даних LIFO (Last-In-First-Out), яка зберігає стан навігації. При виклику команди 'Push View' новий React-компонент монтується поверх існуючого, блокуючи йому доступ до перехоплення клавіш, але зберігаючи його стан у пам'яті. При виклику 'Pop View' верхній екран демонтується, а фокус автоматично повертається попередньому екрану. Така архітектура забезпечує миттєві переходи без втрати введених користувачем даних у попередніх формах.

Найбільшим та найскладнішим є модуль 'files'. Він виступає серцем системи управління контекстом. Модуль містить логіку сканування локальних директорій у пошуках Markdown-файлів, розпізнавання їхньої структури та видобування метаданих (YAML Frontmatter). Завдяки імплементації Drizzle ORM, модуль 'files' зберігає оптимізовані індекси файлів у таблицях SQLite. Кожен файл отримує криптографічний хеш, який дозволяє системі швидко розпізнавати зміни, зроблені зовнішніми редакторами (наприклад, VS Code). Компоненти інтерфейсу цього модуля (List View, Detail View) підключені до глобального стану Zustand, що дозволяє миттєво відображати зміни у файловій системі.

Модуль 'env-settings' відповідає за критичний аспект безпеки - управління конфігураціями та секретними ключами (API Keys). Робота з великими мовними моделями вимагає авторизації. Зберігання ключів безпосередньо у коді є грубим порушенням безпеки. Модуль реалізує парсинг файлів '.env', дозволяє користувачу

безпечно додавати, редагувати або видаляти змінні оточення безпосередньо через TUI. Під час виконання запитів до LLM ці змінні безпечно інжектуються у HTTP-заголовки (Headers), не залишаючи слідів на диску або у журналах логування.

Модуль 'gdrive' вирішує проблему синхронізації та резервного копіювання. Робоче середовище розробника часто охоплює кілька комп'ютерів (наприклад, робочий ноутбук та домашній ПК). Модуль інтегрує офіційний Google Drive API. Логіка реалізації передбачає проходження процесу авторизації OAuth 2.0 (через генерацію токенів доступу та токенів оновлення - Refresh Tokens). Система може автоматично синхронізувати локальну базу промптів із виділеною папкою у хмарі, вирішуючи конфлікти версій на основі міток часу (Timestamps).

Найбільш інноваційним є модуль 'mcp' (Model Context Protocol). Цей модуль реалізує сучасний стандарт комунікації між інструментами та AI-моделями, розроблений компанією Anthropic. Програмна реалізація цього модуля дозволяє системі експортувати власні локальні файли або функції (наприклад, читання вмісту git-репозиторію) як інструменти (Tools), які мовна модель може викликати автономно (Function Calling). Модуль 'mcp' містить складні парсери JSON-схем, які транслюють можливості системи у зрозумілий для AI формат. Це перетворює систему з пасивного сховища тексту на активного AI-агента (Agentic AI).

Важливою складовою програмної реалізації є система обробки помилок та відновлення (Error Handling and Recovery). У термінальному середовищі необроблена виключна ситуація (Unhandled Exception) призводить до негайного аварійного завершення процесу, що може спричинити втрату незбережених даних або пошкодження файлу бази даних SQLite. Для запобігання цьому реалізовано глобальний перехоплювач помилок (Global Error Boundary), який ловить виключення на рівні React-дерева та на рівні процесу Node.js/Bun. При виникненні помилки система виводить інформативне повідомлення у StatusBar, зберігає стек виклику (Stack Trace) у лог-файл та продовжує роботу без падіння.

Надзвичайно критичним аспектом продуктивності TUI-додатків є оптимізація споживання оперативної пам'яті (Memory Optimization) та управління збирачем сміття (Garbage Collection, GC). Оскільки React постійно генерує нові об'єкти під час

фази реконсиляції (Reconciliation), рушій JavaScriptCore повинен ефективно звільняти пам'ять від непотрібних вузлів. Для запобігання 'зупинкам світу' (Stop-the-world GC Pauses), які призводять до візуальних 'фрізів' в інтерфейсі терміналу, розроблено архітектуру агресивного пулінгу об'єктів (Object Pooling). Замість створення нових масивів та об'єктів під час малювання екрану, система перевикористовує існуючі комірки пам'яті. Також ретельно контролюються підписки на події клавіатури, гарантуючи обов'язковий виклик методу 'removeListener' при розмонтуванні компонентів, що унеможливлює витоки пам'яті (Memory Leaks), характерні для довготривалих сесій.

Для забезпечення високої надійності та стійкості кодової бази впроваджено жорсткі стандарти автоматизованого тестування. Система тестування поділена на дві категорії: модульні тести (Unit Tests) та наскрізні тести (E2E Tests). Модульне тестування реалізовано за допомогою фреймворку Bun:test (який є drop-in заміною для Jest/Vitest). Кожен обробник подій Mediator, кожен клас валідації YAML та кожен сервіс DI-контейнера покриті тестами з ізоляцією залежностей через мок-функції (Mock Functions). Це гарантує, що рефакторинг однієї частини системи не зламає іншу.

Наскрізне тестування (End-to-End, E2E) для TUI-додатків є складним викликом, оскільки відсутній браузер і такі інструменти як Cypress чи Playwright не працюють. Для вирішення цієї проблеми було створено кастомний фреймворк тестування терміналу. Він запускає систему у віртуальному PTY (Pseudo-Terminal), програмно надсилає сирі ANSI-послідовності натискання клавіш (наприклад, симуляцію натискання клавіші Enter) та аналізує stdout на наявність очікуваних фрагментів тексту. Це дозволяє повністю автоматизувати перевірку складних сценаріїв користувача (наприклад, 'створити файл -> відредагувати -> зберегти -> знайти в списку').

Окремо слід зазначити реалізацію системи гарячих клавіш (Keybinding System). На відміну від веб-додатків, де комбінації клавіш обробляються через стандартний addEventListener, у TUI кожна клавіша є послідовністю байтів у потоці stdin. Система реалізує централізований реєстр клавіатурних скорочень, де кожен

модуль може зареєструвати свої унікальні комбінації. При натисканні клавіші F1 система відображає оверлей з усіма доступними комбінаціями для поточного екрану, що значно покращує зручність використання (Discoverability).

3.4. Керівництво користувача

Даний розділ містить вичерпні інструкції щодо розгортання, налаштування та повсякденного використання розробленої інформаційної системи. Завдяки концепції Zero-Dependency (відсутність зовнішніх залежностей типу Docker чи серверів БД), процес встановлення є максимально оптимізованим.

Крок 1. Підготовка середовища та встановлення. Перед запуском системи необхідно переконатися, що в операційній системі встановлено середовище виконання Bun (мінімальна версія 1.0). Користувачі Linux та macOS можуть встановити його за допомогою офіційного скрипта з сайту bun.sh, а користувачі Windows - через підсистему WSL 2 або офіційний інсталятор. Далі необхідно схилити (clone) репозиторій з вихідним кодом проєкту за допомогою команди 'git clone'. Після цього користувач повинен перейти у директорію проєкту та виконати команду 'bun install'. Ця команда ініціює завантаження всіх необхідних залежностей (зокрема @opentui/react, react, drizzle-orm, sqlite) з репозиторію пакетів, що завдяки оптимізаціям Bun займає лічені секунди.

Крок 2. Налаштування бази даних та міграції. Оскільки система використовує Drizzle ORM для роботи з SQLite, перед першим запуском необхідно згенерувати та застосувати міграції бази даних. Міграції гарантують, що локальний файл бази даних (.sqlite) міститиме правильну структуру таблиць для модулів 'files' та 'mcp'. Користувач повинен виконати команду 'bun run db:push' (або відповідний скрипт, налаштований у package.json). Ця команда атомарно створить потрібні таблиці та індекси на жорсткому диску.

Крок 3. Налаштування змінних оточення. У кореневій директорії проєкту необхідно створити файл '.env' (або скопіювати з '.env.example'). За допомогою будь-якого текстового редактора користувач повинен внести туди свої персональні API-ключі. Наприклад: OPENAI_API_KEY=sk-..., ANTHROPIC_API_KEY=sk-ant-...,

WORKSPACE_PATH=/absolute/path/to/prompts. Правильне налаштування WORKSPACE_PATH є критичним, оскільки воно вказує системі, яку саме директорію на комп'ютері слід індексувати та сканувати на наявність Markdown-шаблонів.

Крок 4. Запуск системи та інтерфейс. Запуск основного додатку здійснюється командою 'bun run start'. За лічені мілісекунди вікно терміналу очищується, і на екран виводиться головне меню системи (Dashboard), побудоване на базі компонентів OpenTUI. Інтерфейс поділений на кілька логічних панелей (Panels): навігаційне меню (зліва), панель списку файлів/шаблонів (посередині) та панель попереднього перегляду або редагування (справа). У нижній частині екрану завжди відображається панель стану (Status Bar) з підказками гарячих клавіш.

Крок 5. Навігація та взаємодія (Гарячі клавіші). Однією з головних переваг TUI є відмова від використання комп'ютерної миші (Mouse-less Navigation), що значно пришвидшує роботу досвідчених користувачів. Навігація між елементами списків та меню здійснюється за допомогою клавіш 'Стрілка вгору' та 'Стрілка вниз'. Для підтвердження вибору (наприклад, відкриття шаблону або входу в меню) використовується клавіша 'Enter'. Для повернення на попередній рівень ієрархії, закриття модальних вікон (Pop-ups) або відміни дій використовується клавіша 'Escape'. Роутер системи миттєво відреагує на ці команди, оновивши компоненти на екрані.

Крок 6. Створення та редагування шаблонів інструкцій. Для створення нового промпту користувач повинен перейти у модуль керування файлами ('Files Menu'). Після цього необхідно натиснути гарячу клавішу 'F2'. Ця дія активує форму введення (Input Component) та перемістить курсор терміналу у текстове поле. Користувачу буде запропоновано ввести назву нового файлу. Після підтвердження ('Enter'), система створить новий .md файл на диску, згенерує базовий блок метаданих (YAML Frontmatter) та відкриє його у режимі редагування.

Крок 7. Робота з буфером обміну (Clipboard Integration). Редагування великих текстів у терміналі може бути складним завданням. Проте розроблена система тісно інтегрована з системним буфером обміну операційної системи. Якщо користувач

скопіював великий фрагмент коду зі своєї IDE (VS Code, IntelliJ), він може вставити його в редактор TUI за допомогою стандартної комбінації клавіш 'Ctrl+V' (або 'Cmd+V' на macOS). Під капотом система перехоплює цю комбінацію та виконує системний виклик до 'powershell.exe' (на Windows) або 'pbpaste' / 'xclip' (на Unix-системах), зчитує вміст буфера та миттєво інжектує його у внутрішній стан компонента <input> чи <text> (через функцію instance.insertText()). Це дозволяє уникати обмежень стандартного подієвого циклу терміналу при вставці великих масивів даних.

Крок 8. Завершення роботи. Для безпечного завершення роботи з програмою користувач може натиснути комбінацію 'Ctrl+C'. Ядро системи перехопить сигнал операційної системи (SIGINT), коректно закриє з'єднання з базою даних SQLite (уникнувши пошкодження файлу .sqlite), очистить екран терміналу (очищення буфера) і поверне керування стандартній командній оболонці.

Крок 9. Робота з модулем Google Drive. Для активації хмарної синхронізації користувач має перейти до модуля 'gdrive' через головне навігаційне меню. При першому запуску система ініціює процедуру авторизації OAuth 2.0: відкриє посилання для аутентифікації, після чого користувач повинен надати дозвіл на доступ до файлів Google Drive. Після успішної авторизації система збереже токени оновлення (Refresh Tokens) та автоматично синхронізуватиме локальну директорію промптів із виділеною папкою у хмарному сховищі. Конфлікти версій вирішуються автоматично на основі порівняння часових міток останньої модифікації файлів.

Крок 10. Робота з модулем MCP. Модуль Model Context Protocol доступний через окремий пункт навігаційного меню. Він відображає список зареєстрованих MCP-інструментів (Tools) та їхні JSON-схеми параметрів. Користувач може переглянути, які саме локальні функції системи експортуються як інструменти для AI-моделей, та налаштувати параметри їхнього виклику. Це дозволяє системі виступати як MCP-сервер, надаючи мовним моделям стандартизований доступ до локальних ресурсів розробника.

Крок 11. Створення кастомних MCP-інструментів (Advanced Workflow). Якщо розробник бажає інтегрувати свою власну базу даних або корпоративне API у

робочий процес AI, він може визначити новий MCP-інструмент безпосередньо через TUI. За допомогою клавіші 'Ins' (Insert) у модулі MCP відкривається форма створення нової схеми інструменту. Користувач повинен описати ім'я функції, її короткий опис (який AI-модель використовуватиме для прийняття рішення про її виклик) та строгу JSON Schema необхідних аргументів. Збережений інструмент автоматично стає доступним усім підключеним локальним агентам, що дозволяє AI автономно виконувати складні запити в межах корпоративного середовища.

Висновки до розділу 3

У третьому розділі кваліфікаційної роботи було проведено вичерпне обґрунтування вибору програмного стека та детальне дослідження програмної реалізації розробленої системи. Обґрунтовано доцільність використання сучасного, високопродуктивного середовища виконання Bun у комбінації з мовою TypeScript, бібліотекою React 19 та консольним рендерером OpenTUI. Цей стек дозволив перенести парадигму декларативного створення інтерфейсів з браузерного середовища у термінал, забезпечивши безпрецедентну швидкість та низьке споживання ресурсів.

Визначено та класифіковано вимоги до апаратного та програмного забезпечення. Показано, що завдяки оптимізаціям та відмові від важких графічних рушіїв, система здатна ефективно функціонувати навіть на обладнанні базового рівня (від 128 МБ RAM), залишаючись при цьому кросплатформною (Windows, macOS, Linux).

Глибокий архітектурний аналіз підтвердив ефективність використання принципів Atomic Modular Architecture. Детально розібрано роботу інфраструктурного ядра системи: створення власного Dependency Injection контейнера (з різними життєвими циклами), реалізацію патерну Mediator для декуплінгу бізнес-логіки від візуального інтерфейсу, та функціонування динамічного проксі-роутера. Також детально розкрито логіку ключових робочих модулів: 'files' (робота з файловою системою та Drizzle ORM/SQLite), 'env-settings' (управління безпекою), 'gdrive' (хмарна синхронізація) та 'mcp' (Model Context Protocol).

На завершення розроблено покрокове та детальне керівництво користувача, яке охоплює всі етапи роботи з інформаційною системою: від встановлення та налаштування змінних оточення до використання гарячих клавіш (F2, Ctrl+V, стрілки), взаємодії з системним буфером обміну та безпечного завершення процесів. Це керівництво забезпечує повне розуміння життєвого циклу користувацької сесії в межах реактивного термінального середовища.

ЗАГАЛЬНІ ВИСНОВКИ

У кваліфікаційній роботі вирішено актуальну науково-практичну задачу - розроблено локальну, високопродуктивну інформаційну систему централізованого управління контекстом та інструкціями для AI-систем у середовищі терміналу (TUI).

У ході виконання роботи було отримано наступні результати:

1. Проведено системний аналіз предметної області та виявлено ключові проблеми, з якими стикаються розробники при інтеграції великих мовних моделей (LLM). Встановлено, що жорстке кодування інструкцій (Hardcoding), відсутність версійності та фрагментація контексту призводять до феномену дрейфу інструкцій (Prompt Drift) та значно знижують ефективність роботи. Обґрунтовано необхідність переходу до парадигми Prompt-as-Code та використання реактивних термінальних інтерфейсів (TUI) як оптимального рішення, що усуває проблеми контекстного перемикання, притаманні важким хмарним SaaS-платформам.

2. Спроєктовано та обґрунтовано гібридну архітектуру збереження даних. Для забезпечення максимальної портативності та гнучкості фізичне зберігання шаблонів реалізовано на базі нативної файлової системи (Markdown-файли з YAML Frontmatter). Водночас для гарантування миттєвого повнотекстового пошуку розроблено шар кешування на базі In-Memory бази даних SQLite, взаємодія з якою абстрагована за допомогою типізованого інструменту Drizzle ORM. Розроблений механізм хешування файлів забезпечує абсолютну консистентність даних між диском та TUI-інтерфейсом.

3. Розроблено складне математичне та алгоритмічне забезпечення, яке дозволило реалізувати реактивний інтерфейс у текстовому терміналі. Успішно імплементовано алгоритм диференціального оновлення дерев (Reconciliation) бібліотеки React 19 для термінальних вузлів (<box>, <text>). Застосування алгебраїчних моделей Flexbox-компонування (у ядрі на базі Zig), алгоритмів віртуальної прокрутки (Virtual Scrolling) та кінцевих автоматів (FSM) для розпізнавання багатобайтових клавіатурних ANSI-послідовностей гарантувало

стабільну частоту оновлення екрану (60 FPS) незалежно від обсягу оброблюваного тексту.

4. Створено масштабовану програмну реалізацію на базі середовища виконання Bun, яка базується на принципах Atomic Modular Architecture. Розроблено кастомний Dependency Injection (DI) контейнер з підтримкою життєвих циклів (Singleton, Scoped, Transient) та впроваджено архітектурний патерн Mediator (CQRS-lite). Це забезпечило повний декуплінг бізнес-логіки робочих модулів ('files', 'env-settings', 'gdrive', 'mcp') від візуального відображення та управління станом через бібліотеку Zustand.

5. Забезпечено інтеграцію сучасного стандарту Model Context Protocol (MCP), що перетворює створену систему з пасивного текстового редактора на активну агентну платформу, здатну динамічно надавати мовним моделям локальний контекст. Написано детальне керівництво користувача, що покриває весь життєвий цикл використання додатка, включно з управлінням через 'гарячі' клавіші та системною інтеграцією з буфером обміну операційної системи.

Практичне значення одержаних результатів полягає у тому, що створена система повністю готова до експлуатації та здатна суттєво пришвидшити процеси розробки, тестування та інтеграції систем штучного інтелекту, вирішуючи проблему фрагментації промптів. Розроблений інструментарій відзначається нульовою залежністю від зовнішніх серверів (Zero-Dependency), мінімальними вимогами до ресурсів (від 128 МБ RAM) та високим рівнем безпеки управління конфігураціями.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Офіційна документація середовища виконання Bun. URL: https://bun.sh/docs/
2. Документація бібліотеки React 19 та архітектури React Fiber. URL: https://react.dev/
3. TypeScript Handbook. Офіційний посібник розробника Microsoft. URL: https://www.typescriptlang.org/docs/
4. Drizzle ORM Documentation. Type-safe ORM for TypeScript. URL: https://orm.drizzle.team/
5. Офіційна специфікація Model Context Protocol (MCP) від Anthropic. URL: https://modelcontextprotocol.io/
6. Документація бази даних SQLite. Architecture of SQLite. URL: https://www.sqlite.org/arch.html
7. Управління станом додатків за допомогою Zustand. URL: https://github.com/pmndrs/zustand
8. Microsoft AutoGen Documentation. Офіційна документація передового фреймворку для створення мультиагентних систем, де агенти взаємодіють між собою для вирішення задач. URL: https://microsoft.github.io/autogen/
9. CrewAI Documentation. Практична документація популярного фреймворку для оркестрації автономних AI-агентів із заданими ролями та інструкціями. URL: https://docs.crewai.com/
10. LangGraph Documentation. Бібліотека для створення stateful, мультиагентних застосунків з LLM від творців LangChain. URL: https://langchain-ai.github.io/langgraph/
11. Vercel AI SDK. Потужний інструментарій (Toolkit) для інтеграції AI в React та TypeScript застосунки (дуже релевантно для твоєї архітектури на React 19). URL: https://sdk.vercel.ai/docs
12. Microsoft Semantic Kernel. Документація SDK для розробників, що дозволяє легко інтегрувати AI-агентів та плагіни в існуючий код. URL: https://learn.microsoft.com/en-us/semantic-kernel/

13. LlamaIndex Docs. Фреймворк даних для підключення кастомних джерел даних до LLM (критично важливо для RAG та підготовки контексту). URL: https://docs.llamaindex.ai/en/stable/
14. Model Context Protocol (MCP) Official GitHub Repositories. Специфікації та SDK реалізації стандарту MCP для взаємодії моделей з локальними інструментами. URL: https://github.com/modelcontextprotocol
15. OpenAI Cookbook. Відкритий репозиторій з практичними прикладами коду, патернами створення агентів та управлінням контекстом через API. URL: https://cookbook.openai.com/
16. OpenAI Vision API Guide. Офіційна документація щодо того, як GPT-4o розпізнає зображення, обчислює токени для картинок (High/Low resolution) та розуміє контекст. URL: https://platform.openai.com/docs/guides/vision
17. Anthropic Vision API (Claude 3). Практичний посібник з роботи з мультимодальністю в моделях Claude, включаючи найкращі практики промптингу для аналізу графіків та схем. URL: https://docs.anthropic.com/en/docs/build-with-claude/vision
18. Google Gemini Vision Documentation. Документація по роботі з Gemini API для аналізу зображень, відео та аудіо з прикладами коду. URL: https://ai.google.dev/gemini-api/docs/vision
19. Парсинг документів та зображень у Markdown (Must-have для RAG):
20. LlamaParse (by LlamaIndex). Офіційний репозиторій топового інструменту на базі AI, який спеціалізується на парсингу складних документів (з таблицями, картинками та графіками) у чистий Markdown, що ідеально підходить для контексту LLM. URL: https://github.com/run-llama/llama_parse
21. Marker (by Vik Paruchuri). Високопродуктивний open-source конвеєр моделей глибокого навчання для швидкого перетворення PDF та зображень у Markdown із високою точністю. URL: https://github.com/VikParuchuri/marker
22. Unstructured.io Documentation. Індустріальний стандарт для препроцесингу та парсингу неструктурованих даних (зображень, сканів, PDF) для систем RAG (Retrieval-Augmented Generation). URL: https://docs.unstructured.io/

23. Відкриті мультимодальні моделі (Open-Source Vision LLMs):
24. LLaVA (Large Language-and-Vision Assistant). Документація та сторінка проєкту однієї з найвідоміших відкритих мультимодальних моделей, яка поєднує візуальний енкодер CLIP із мовною моделлю LLaMA. URL: https://llava-vl.github.io/
25. Qwen-VL (Alibaba Cloud). Репозиторій та опис передової відкритої моделі для розпізнавання зображень, читання тексту на картинках (OCR) та візуального відповідання на запитання. URL: https://qwenlm.github.io/blog/qwen-vl/
26. Microsoft Florence-2. Огляд легкої, але надзвичайно потужної моделі від Microsoft, яка вміє робити OCR, детекцію об'єктів та сегментацію зображень на основі текстових промптів. URL: https://huggingface.co/blog/finetune-florence2
27. Архітектура візуальних трансформерів та нові підходи:
28. ColPali: Visual Retrievers for RAG. Стаття від Hugging Face про революційний підхід до RAG, де замість парсингу тексту з документа у вектори, документ розбивається на зображення і аналізується мультимодальною моделлю напрямую. URL: https://huggingface.co/blog/colpali
29. A Guide to Vision-Language Models. Детальний інженерний гайд від Hugging Face щодо того, як саме працюють моделі "віжну" (Vision Transformers) у зв'язці з текстовими моделями. URL: https://huggingface.co/blog/vision-language-models
30. Hugging Face Computer Vision Course. Відкритий практичний курс з комп'ютерного зору, що покриває базові архітектури ViT (Vision Transformers), які лежать в основі сучасних мультимодальних LLM. URL: https://huggingface.co/learn/computer-vision-course/